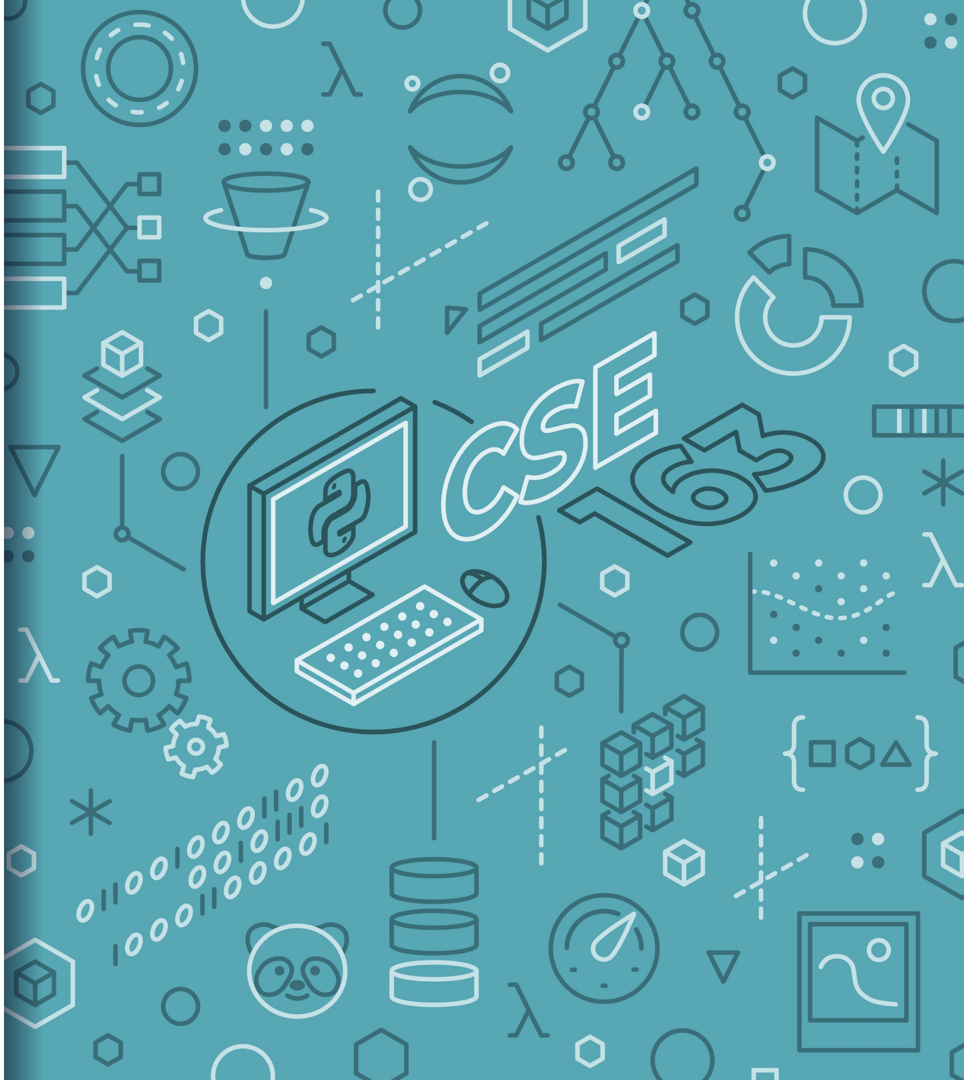
CSE 163

Introduction to Efficiency

Nicole Riley



Objectives

- Understand the difference between timing versus steps to measure efficiency.
- Why efficiency matters
- Understand meaning of Big O notation.
- Beginning of data structure efficiency.

Efficient code

Two ways to evaluate efficiency:

- Speed (time)
- Space (memory)

There are often trade-offs between the two of them.

We will be focusing on speed.

Why does this matter?

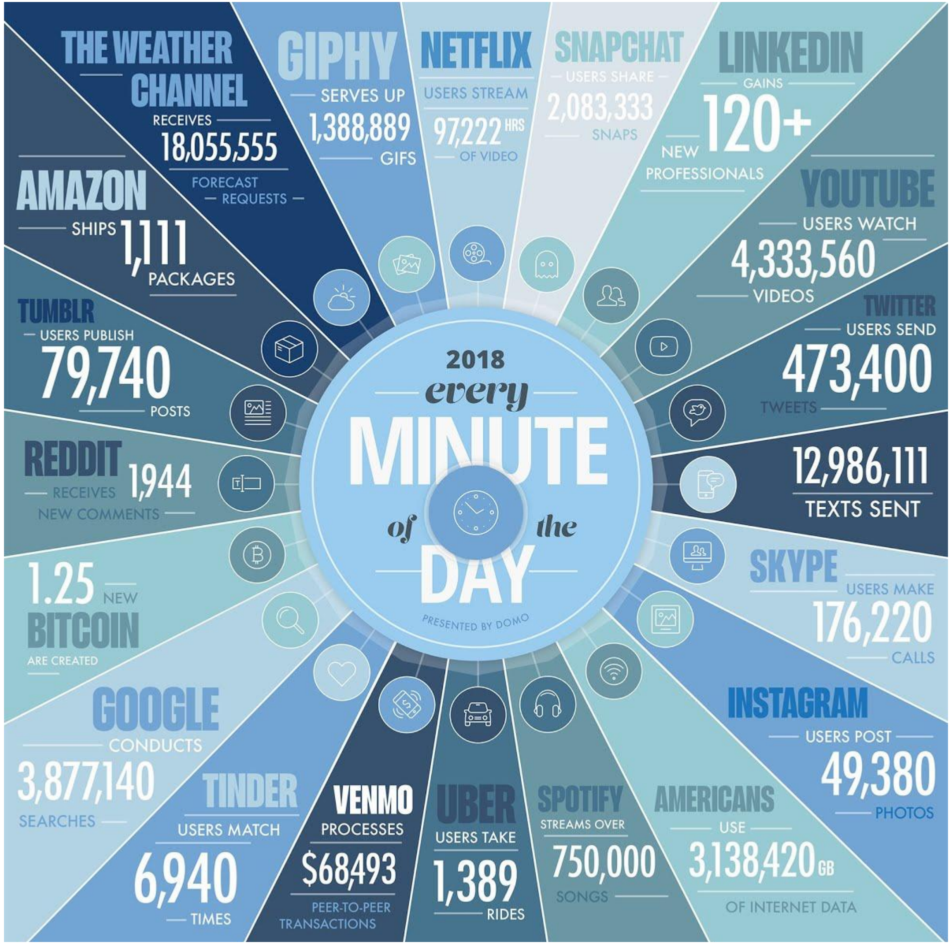
The runtime of programs seems unimportant if you think about small inputs.

But we are focused on data in this class.

If we want to start thinking about analyzing large amounts of data, we care about how fast an algorithm is.

If we choose an algorithm that is too slow, it may greatly impede the speed of our analysis.

Why does this matter?



https://www.digitalinformationworld.com/2018/06/infographics-data-never-sleeps-6.html Domo

Evaluating runtime

```python
# assumes n is larger than 1. n is
# inclusive and an integer
def sum_of_squares(n):
  total = 0
  for i in range(1, n + 1):
    total += i ** 2
  return total


def sum_of_squares2(n):
  return (n * (n + 1) * (2*n + 1)) // 6
```

Which one of these functions is faster? How to we measure this?

One way: Time

Demo

How fast do our functions run when:

```
n = 10
n = 1000
n = 1000000
```

Problems

- Runtimes can vary between computers
- Runtimes can vary on the same machine.

Method 2: Count steps

Assumption: A single simple line of code takes the same amount of time to run in Python.

Examples:

```
num = 5 + 10
print("hello")
num > 5 and num % 2 == 0
```

Rules:

- Loop runtime is number of times it is run (N) times the number of statements
- Method runtime is sum of all statements found inside of it.

Examples

```
def example_method(n):
  statement1
  statement2
  statement3

  for i in range(n):
    statement4

  for i in range(n):
    statement5
    statement6
    statement7
```

$4n + 3$

Examples

```
def example_method2(n):

  for i in range(n):
    for j in range(n):
      statement

  statement
```

$1 + n^2$

sum_of_squares

```python
# assumes n is larger than 1. n is
# inclusive and an integer
def sum_of_squares(n):
  total = 0
  for i in range(1, n + 1):
    total += i ** 2
  return total

def sum_of_squares2(n):
  return (n * (n + 1) * (2*n + 1)) // 6
```

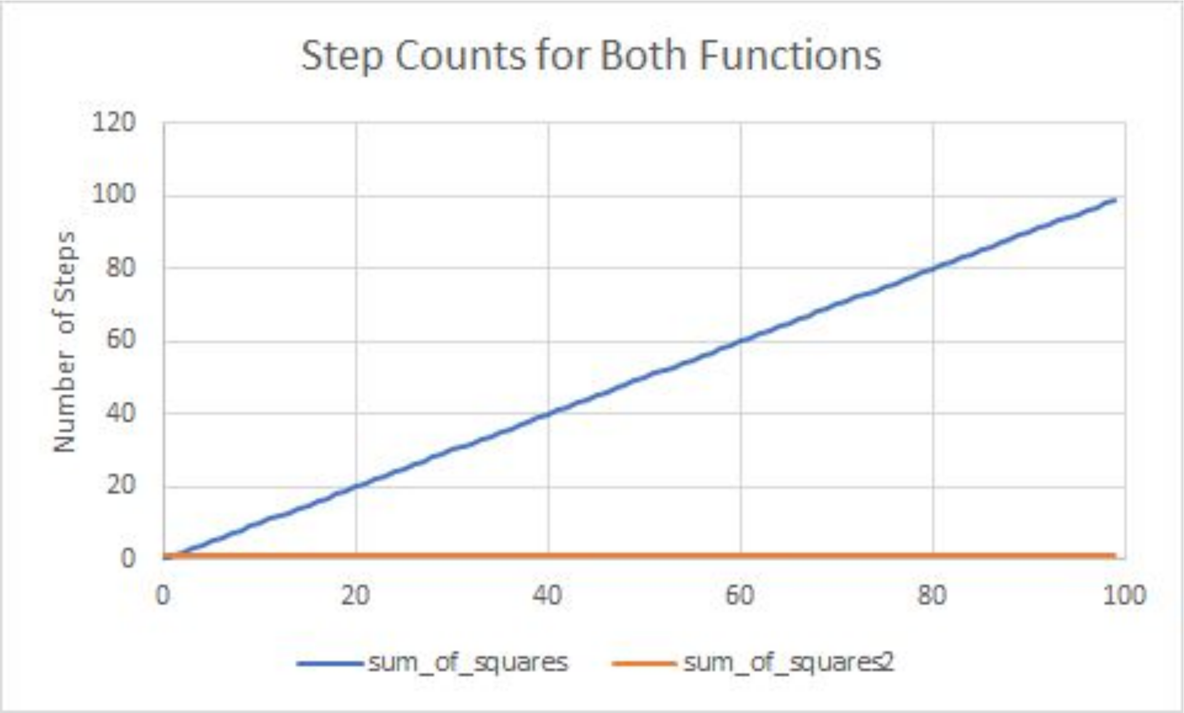
n + 2

1

Two functions differ in terms of runtime by an order of magnitude (n).

How does this look as input increases in size?

Visual


Step Counts for Both Functions
Number of Steps
120
100
80
60
40
20
0
0
20
40
60
80
100
sum_of_squares
sum_of_squares2

Growth rate

We often think of algorithms in terms of growth rate in terms of the input or input data size (**n**)

Think of the runtime of $5n^3 + 4n^2 + 3n + 7$.

All terms seem significant when we look at smaller inputs.

However, what happens when n becomes extremely large? At that point the term with the largest power of n will dominate (n^3).

We say it runs on the order of n^3 or $O(n^3)$ (**Big Oh of n cubed**).

Poll Everywhere

Think

1 min

pollev.com/cse163

What is the runtime of method1?

```
def method1(n):
  value = 0
  for i in range(9):
    for j in range(n):
      value += 7 * i + j
  return value ** 2
```

A. $O(n)$
B. $O(9n + 2)$
C. $O(n^2)$
D. $O(n^3)$
E. $O(9n)$

Poll Everywhere

Pair

1 min

pollev.com/cse163

What is the runtime of method1?

```
def method1(n):
  value = 0
  for i in range(9):
    for j in range(n):
      value += 7 * i + j
  return value ** 2
```

A. $O(n)$
B. $O(9n + 2)$
C. $O(n^2)$
D. $O(n^3)$
E. $O(9n)$

Small Change

How does the runtime change with a small change?

```
def method1(n):
  value = 0
  for i in range(9):
    for j in range(n):
      value += 7 * i
      value += j
  return value ** 2
```

Does the difference between **18n** and **9n** matter?

Coefficients

Why are constants and coefficients less important?

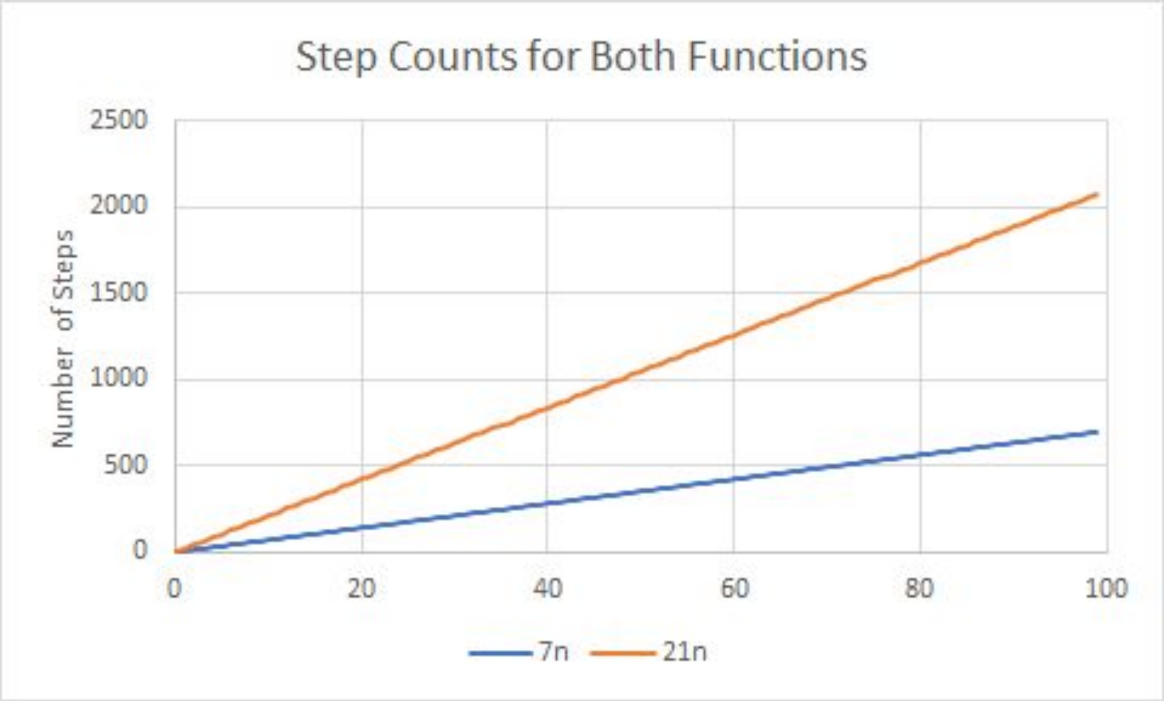
What is the relative difference between an algorithm with a runtime of

7n

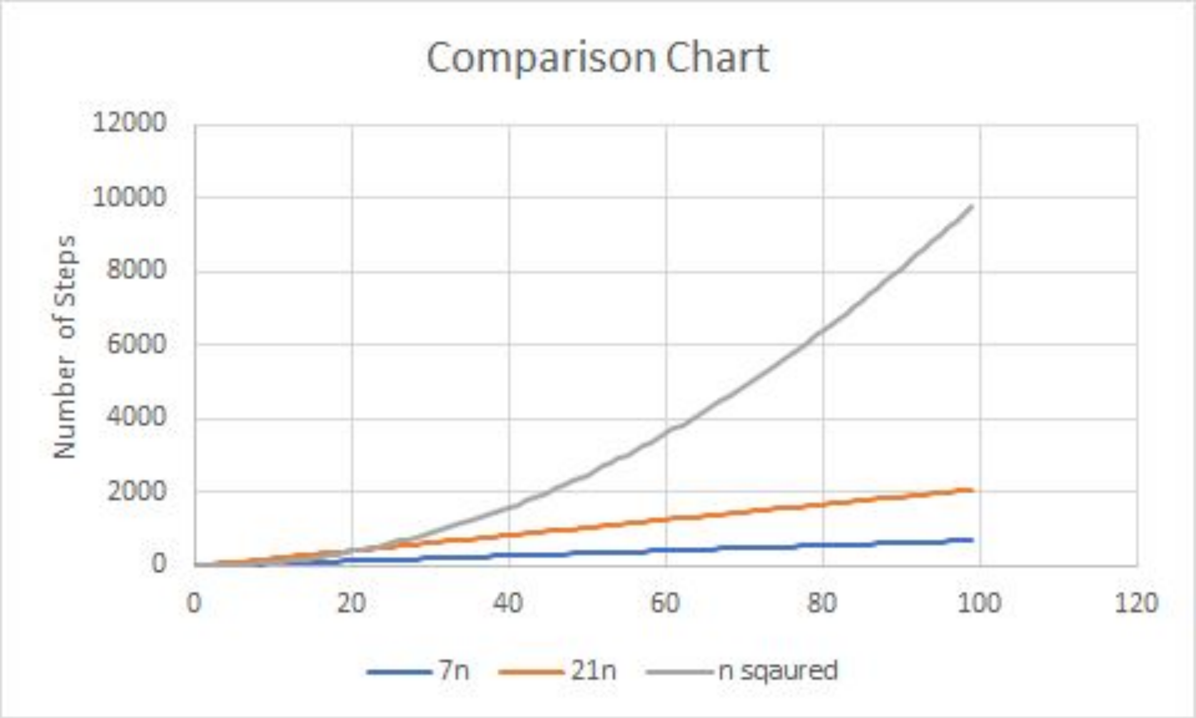
versus

21n

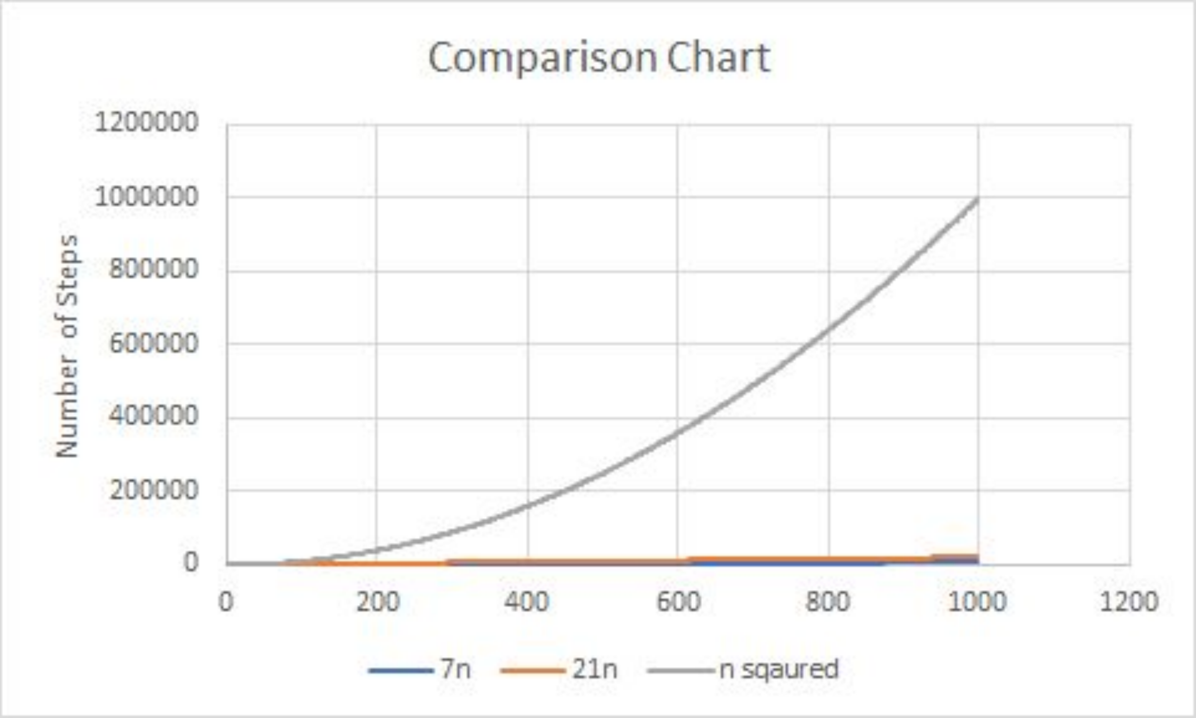
Visual


Step Counts for Both Functions
Number of Steps
2500
2000
1500
1000
500
0
0
20
40
60
80
100
7n
21n

Quadratic Comparison



Quadratic Comparison

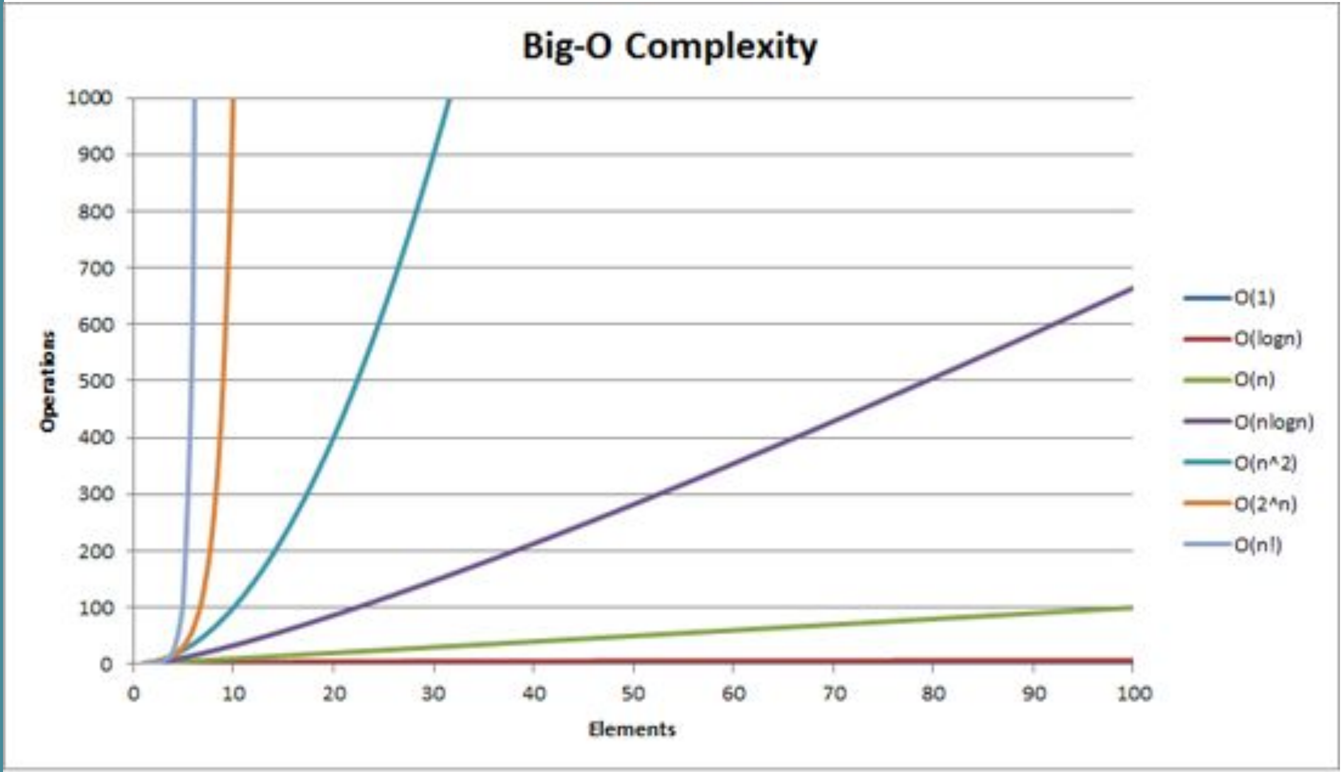


Complexity classes

A category of algorithm efficiency based on the algorithm's relationship to the input size n

Class	Constant	If n doubles, then
Constant	$O(1)$	unchanged
Linear	$O(n)$	doubles
Quadratic	$O(n^2)$	quadruples
Cubic	$O(n^3)$	Multiplies by 8
...		
Exponential	$O(2^n)$	Multiplies drastically

Complexity classes



http://recursive-design.com/blog/2010/12/07/comp-sci-101-big-o-notation/ - post about a Google interview

Brain Break

Data Structure Efficiency

Demo

Because we care about the speed our code runs, we also care about the data structures we use to complete tasks.

Today we will be focusing on two python data structures

- Lists
- Sets

List Solution

```
def count_unique_list(file_name):
  words = list()
  with open(file_name) as file:
    for line in file.readlines():
      for word in line.split():
        if word not in words:
          words.append(word)
  return len(words)
```

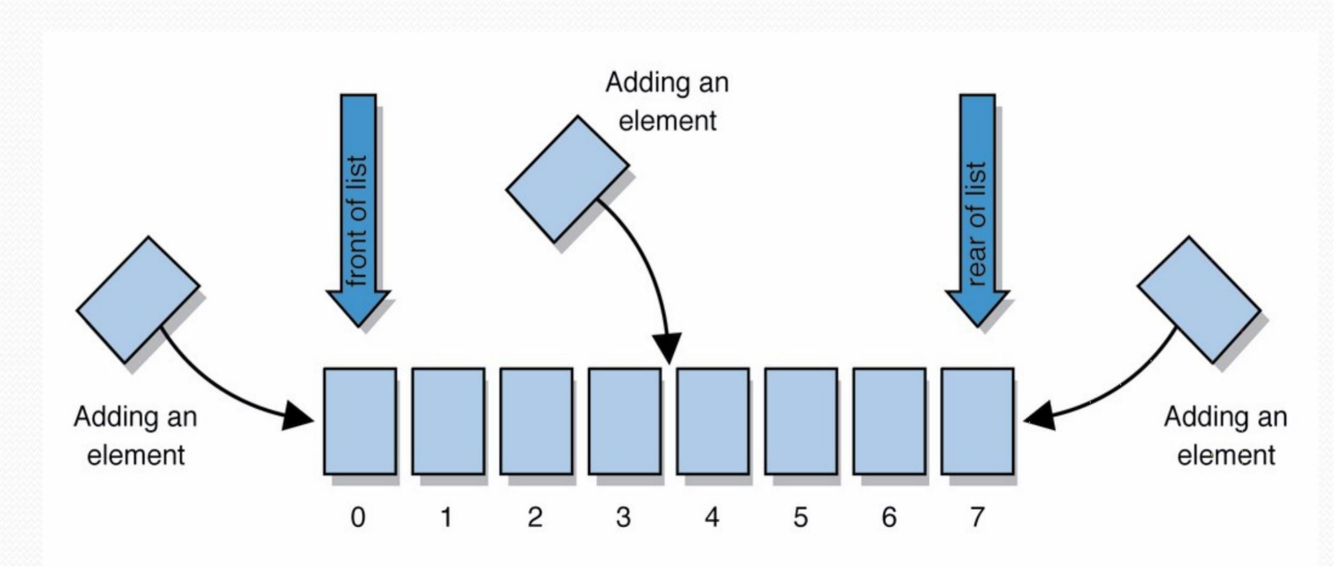
Set Solution

```
def count_unique_set(file_name):
  words = set()
  with open(file_name) as file:
    for line in file.readlines():
      for word in line.split():
        if word not in words:
          words.add(word)
  return len(words)
```

Implementing the in operator

On a high level, what do we need to do to implement in for a list?

Let's assume list elements are stored as smaller elements in a single contiguous piece of memory



Implementing the in operator

How long would this operation take?

Well it depends on both the list and the word

What if we are looking for the word “I”?

What if we are looking for the word “dogs”?

What if we are looking for the word “corgi”?

We can think about our worst case input: when the word is not in the list.

```
list_example = [“I”, “like”, “petting”, “dogs”]
```

Difference in Efficiency

Why does a set or list make a difference in terms of efficiency?

Lists and sets serve different purposes and have different operations optimized (meant to be faster).

in operation is what makes the difference.

Searching in a list is O(n)

Searching in a set is O(1)

```
list_example = [“I”, “like”, “petting”, “dogs”]
set_example = {“I”, “like”, “petting”, “dogs”}
```

Poll Everywhere

Think

1 min

pollev.com/cse163

What is the runtime of method2?

```
def method2(n):
  lst = [];
  for i in range(n):
    lst.append([j ** 2 for j in range(n) if j % 2 == 0])
  return lst
```

A. $O(n)$
B. $O(1)$
C. $O(n^2)$
D. $O(n^3)$
E. $O(2n^2)$

Poll Everywhere

Pair

1 min

pollev.com/cse163

What is the runtime of method2?

```
def method2(n):
  lst = [];
  for i in range(n):
    lst.append([j ** 2 for j in range(n) if j % 2 == 0])
  return lst
```

A. $O(n)$
B. $O(1)$
C. $O(n^2)$
D. $O(n^3)$
E. $O(2n^2)$